# Building Winning Trading Systems with TradeStation™

**George Pruitt**
**John R. Hill**
FUTURES TRUTH

**John Wiley & Sons, Inc.**

# Building Winning Trading Systems with TradeStation™

Founded in 1807, John Wiley & Sons is the oldest independent publishing company in the United States. With offices in North America, Europe, Australia and Asia, Wiley is globally committed to developing and marketing print and electronic products and services for our customers' professional and personal knowledge and understanding.

The Wiley Trading series features books by traders who have survived the market's ever changing temperament and have prospered—some by reinventing systems, others by getting back to the basics. Whether a novice trader, professional, or somewhere in between, these books will provide the advice and strategies needed to prosper today and well into the future.

For a list of available titles, please visit our Website at www.WileyFinance.com.

# Building Winning Trading Systems with TradeStation™

**George Pruitt**
**John R. Hill**
FUTURES TRUTH

**John Wiley & Sons, Inc.**

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada



***Library of Congress Cataloging-in-Publication Data:***
Pruitt, George, 1967–
Building winning trading systems with TradeStation™ / George Pruitt, John R. Hill.
p. cm. — (the Wiley trading series)
Includes index.
ISBN 0-471-21569-4 (cloth: alk. paper)
1. Investments—Data processing. 2. Stocks—Data processing. I. Hill, John R., 1926– II. Title. III. Series.
HG4515.95.P78 2002
332.64'2'02855369—dc21

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my loving wife and family.
J. H.

I would like to dedicate this book to Mary, Cliff, Butch, and Marilyn for their eternal courage and support. I would also like to thank my loving wife, Leslie and my patient and understanding children, Brandon and Emily.
G. P.

# Contents

# Acknowledgments

We would like to thank TradeStation Securities, OptionVue software, and Jan Arps and Len Yates for their contributions.

# Introduction

Before System Writer (the grandfather of today's current TradeStation), system developers and traders did not have a commercialized software platform to develop their trading ideas. We are not talking about charting packages; we are talking about programs that could understand a trading strategy based on technical analysis. Sure, there were other sophisticated programs in the 1980s that could be used, but they required a thorough understanding of programming, additional software, and extreme patience with the software developer. Most of these programs were not open platform; you had to program within their limited scripting language and could not share your programs with others. The programs that were potentially open platform usually required a separate editor and compiler and a very sophisticated user. In fact, we developed and still use a package that is based on a FORTRAN compiler. The System Writer/TradeStation programs provided one sleek package that gave the ability to test, optimize, and trade to the trading masses. TradeStation has been accepted worldwide as the premier market analysis platform and the standard against which other trading/testing platforms are measured. The latest version of TradeStation has evolved into more than just software; it is a fully self-contained trading platform that incorporates direct access brokerage for futures and equities traders. Along with the birth of TradeStation 6.0, Omega Research has been reborn and renamed to TradeStation Securities.

We are not here to proclaim TradeStation's Securities products to be the best in the business, but we are here to educate their users on how to use what is accepted as the industry standard. Of course Omega Research has had their problems; all software developers have had their problems. Recently, it was heard that the inventor of the CTRL + ALT + DEL key sequence at IBM

stated that he may have created it, but Microsoft made it famous. What sets TradeStation apart from the other software packages is its powerful scripting language EasyLanguage. EasyLanguage is more powerful than easy; it really isn't a scripting language, but more of a full-blown programming language. It can be compared to BASIC, FORTRAN, Pascal, or C. In fact, it is based off of a Pascal compiler and has been around since 1987. Sam Tennis, the father of EasyLanguage, wanted to provide traders with a simple and logical language that required little programming knowledge. In our opinion, he was successful in this endeavor.

Technical analysis of stocks and commodities is complex and you would think that a programming language that deals with such a lofty subject would be as complex. Fortunately, EasyLanguage comes with a complete library of the industry's most widely used analysis techniques. Third-party developers have further extended this library. Traders do not need to recreate the wheel, nor do they need the programming knowledge to put these techniques into action. Traders can even customize their own ideas or existing ones and add them to the library.

This book is designed for all TradeStation and EasyLanguage users. However, this book does expect users to be somewhat familiar with TradeStation and its functions. Beginners can obtain a good foundation on programming technique, program control structures, data structures, and familiarization on the use of the EasyLanguage built-in functions. All users will benefit from the chapters that discuss proper trading system development. We explore all areas of analysis techniques from Indicators to Paint Bars with a special emphasis on trading strategies.

Since a large portion of this book involves actual computer code, a companion CD-ROM, with all of the computer programs and data, is provided for the reader. The analysis techniques are provided in TradeStation 2000i and TradeStation 6.0 formats. Previous version users can still utilize the analysis techniques by simply typing them into their version of PowerEditor. Chapter 5 enlists the help of Microsoft Excel spreadsheet software to create three-dimensional contour charts. Excel isn't necessary, but some type of spreadsheet software is needed to transform the data into a chart.

This book was designed with the trader in mind. Most of the trading techniques were designed and tested with indices, futures, and commodities (we focused on these markets due to their long histories with trading systems). Strictly equity traders will still get a good programming and good system design education. Index (mini or full-size) and futures traders will be privy to five highly successful trading approaches. These approaches are good launching pads for further and much more detailed research.

In our opinion, the best way to make full use of this book is for the reader to work his way through the book starting with Chapter 1. At the end of chapters that have detailed *analysis techniques* (or *computer programs*—the two terms

are interchangeable), we would suggest loading from the CD-ROM, verifying, and running the programs in TradeStation. The concepts of each chapter should be mastered before moving on to the next chapter.

We hope you (the reader) enjoy this book and that it opens your eyes to the power of TradeStation and your own creativity. Good luck and good trading.

# 1

# Fundamentals

## WHAT IS EASYLANGUAGE?

When you code (slang for writing your ideas into a programming language) an analysis technique, you are directing the computer to follow your instructions to the tee. A computer program is nothing but a list of instructions. A computer is obedient and speedy, but it is only as smart as its programmer. In addition, the computer requires that its instructions to be in an exact format. A programmer must follow certain syntax rules.

EasyLanguage is the medium used by traders to convert a trading idea into a form that a computer can understand. Fortunately for nonprogrammers, EasyLanguage is a high level language; it looks like the written English language. It is a compiled language; programs are converted to computer code when the programmer deems necessary. The compiler then checks for syntactical correctness and translates your source code into a program that the computer can understand. If there is a problem, the compiler alerts the programmer and sometimes offers advice on how to fix it. This is different than a translated language, which evaluates every line as it is typed.

All computer languages, including EasyLanguage, have several things in common. They all have:

- Reserved Words. Words that the computer language has set aside for a specific purpose. You can only use these words for their predefined purposes. Using these words for any other purpose may cause severe problems. (See the list of reserved words in Appendix C.)

- Remarks. Words or statements that are completely ignored by the compiler. Remarks are placed in code to help the programmer, or other people who may reuse the code, understand what the program is designed to do. EasyLanguage also utilizes *skip words*. These words are included in a statement to make the programming easier to read. For example, *Buy on next bar at myPrice stop* is the same as *Buy next bar myPrice stop*. The words *on* and *at* are completely ignored. (See the list of skip words in Appendix C.)
- Variables. User-defined words or letters that are used to store information.
- Data Types. Different types of storage; variables are defined by their data types. EasyLanguage has three basic data types: Numeric, Boolean, and String. A variable that is assigned a numeric value, or stored as a number, would be of the Numeric type. A variable that stores a true or false value would be of the Boolean type. Finally, a variable that stores a list of characters would be of the String type.

### *Variables and Data Types*

A programmer must understand how to use variables and their associated data types before they can program anything productive. Let's take a look at a snippet of code.

```
mySum = 4 + 5 + 6;
myAvg = MySum/3;
```

The variables in this code are *mySum* and *myAvg* and they are of the Numeric data type; they are storage places for numbers. EasyLanguage is liberal concerning variable names, but there are a few requirements. A variable name cannot

- start with a number or a period (.)
- be a number
- be more than 20 alphanumeric characters long
- include punctuation other than the period (.) or underscore (_)

| Correct | Incorrect |
|---|---|
| myAvg | 1MyAvg |
| mySum | .sum |
| sum | val+11 |
| val1 | the//sum |
| the.sum | my?sum |
| my_val | 1234 |

Variable naming is up to the style of the individual programmer. EasyLanguage is not case sensitive (you can use upper or lowercase letters in the variable names). (Note: This is our preference—may not be everybody's.) Lowercase letters are preferred for names that only contain one syllable. For variable names that have more than one syllable, we begin the name with a lowercase letter and then capitalize the beginning of each subsequent syllable.

```
sum, avg, total, totalSum, myAvg, avgValue, totalUpSum, totDnAvg
```

Still referring to the previous snippet of code, mySum is assigned the value of 15 and myAvg is assigned the value of 15/3 or 5. If a variable name is created, it must be declared ahead of time. The declaration statement defines the initial value and data type of the variable. The compiler needs to know how much space to reserve in memory for all variables. The following code is a complete EasyLanguage program. (Note: Most of the code that you will see in this book will be particular to EasyLanguage and will probably not work in any other language.)

```
Vars: mySum(0),myAvg(0);
mySum = High + Low + Close ;
myAvg = mySum/3 ;
```

The *Vars:* (or Variables:) statement tells the computer what variables are being declared and initialized. We declare the variables by simply listing them in the Vars statement and initialize them by placing an initial value in parentheses following the variable name. In this case, mySum and myAvg are to be equal to zero. EasyLanguage is smart enough to realize that these variables should be of the Numeric data type, since we initialized them with numbers. Variable names should be self-descriptive and long enough to be meaningful. Which of the following is more self-explanatory?

```
mySum = High+Low+Close;        or      k = High + Low + Close;
myAvg = mySum/3;                       j = k/3;
BuyPt = Close + myAvg;                 l = Close+j;
```

Variables of Boolean and String types are declared in a similar fashion.

```
Vars: myCondition(false),myString("abcdefgh");
```

The variable *myCondition* was initialized to false. The word *false* is a reserved word that has the value of zero. This word cannot be used for any other purpose. The variable *myString* was initialized to abcdefgh. Sometimes you will need to use a variable for temporary purposes and it is difficult to declare and initialize all of your variables ahead of time. In the case of a temporary variable (one that holds a value for a short period of time), EasyLanguage has declared and initialized several variables for your use; value0 through value99 have been predefined and initialized to zero and are ready for usage in your programs. The following is a complete EasyLanguage program:

```
value1 = High + Low + Close;
value2 = (High + Low)/2.0;
```

Notice there isn't a Vars statement. Since value1 and value2 are predefined, the statement isn't needed. You have probably noticed the semicolon (;) at the end of each line of code. The semicolon tells the compiler that we are done with this particular instruction. In programming jargon, instructions are known as *statements*. Statements are made up of expressions, which are made up of constants, variables, operators, functions, and parentheses. Some languages need a termination symbol and others do not. EasyLanguage needs the statement termination symbol. Remember to put a semicolon at the end of each line to prevent a syntax error.

*Inputs* are similar to variables. They follow the same naming protocol and are declared and initialized. However, an input remains constant throughout an analysis technique. An input cannot start a statement (a line of instruction) and cannot be modified within the body of the code. One of the main reasons for using inputs is that you can change input values of applied analysis techniques without having to edit the actual EasyLanguage code. Inputs would be perfect for a moving average indicator. When you plot this indicator on a chart, you simply type in the length of the moving average into the input box of the dialog box. You don't want to have to go back to the moving average source code and change it and then verify it. Also, when used in trading strategies, inputs allow you to optimize your strategies. This is discussed in Chapter 5.

Notice how inputs and variables are declared in similar style.

```
Inputs: length1(10),length2(20),flag(false);

Vars:   myLength1(10),myAvgVal(30);
```

However, notice how they are used differently in coding.

**Variables**

```
myLength1 = myAvgVal + myLength1;  {Correct}
```

**Inputs**

```
length1 = myAvgVal + length1;      {Incorrect}
myLength1 = length1*2;             {Correct}
```

Variables can start a statement and can be assigned another value. Since inputs are constants and cannot be assigned new values, they cannot start a statement.

In a strongly typed language, such as C, Pascal, or C++, if you assign a real value such as 3.1456 to an integer typed variable, the decimal portion is truncated and you end up with the number 3. As we all know, precision is important when it comes to trading, so EasyLanguage includes only one Numeric type. All numbers are stored with a whole and a fractional part. In the old days when CPUs were slow, noninteger arithmetic took too much time and it was advised to use integer variables whenever possible.

## OPERATORS AND EXPRESSIONS

Previously, we discussed statements and how they are made up of expressions. To review, an expression consists of a combination of identifiers, functions, variables, and values, which result in a specific value. Operators are a form of built-in functions and come in two forms: unary and binary. A binary operator requires two operands, whereas a unary operator requires only one. Most of your dealings with operators in EasyLanguage will be of the binary variety. Some of the more popular ones are: + – / * < = > >= <= <> AND OR. These binary operators can be further classified into two more categories: arithmetic and logical.

Expressions come in three forms: arithmetic, logical, and string. The type of operator used determines the type of expression. An arithmetic expression includes + – / *, whereas a logical or Boolean expression includes < = > >= <= <> AND OR.

| **Arithmetic Expressions** | **Logical Expressions** |
|---|---|
| myValue = myValue + 1; | myCondition1 = sum > total; |
| myValue = sum – total; | myCondition1 = sum <> total; |
| myResult = sum*total+20; | cond1 = cond1 AND cond2 |

Arithmetic expressions always result in a number, and logical expressions always result in true or false. True is equivalent to 1, and False is equivalent to 0. String expressions deal with a string of characters. You can assign string values to string variables and compare them.

```
myName1 = "George Pruitt";
myName2 = "John Hill";
cond1 = (myName1 <> myName2);
myName3 = myName1 + " " + myName2;
```

Concatenation occurs when two or more strings are added together. Basically, you create one new string from the two that are being added together.

## PRECEDENCE OF OPERATORS

It is important to understand the concept of precedence of operators. When more than one operator is in an expression, the operator with the higher precedence is evaluated first and so on. This order of evaluation can be modified with the use of parentheses. EasyLanguage's order of precedence is as follows:

1. Parentheses
2. Multiplication or division
3. Addition or subtraction
4. <, >, =, <=, >=, <>
5. AND
6. OR

Here are some expressions and their results:

| | | |
|---|---|---|
| 1. | 20 – 15/5 | equals 17 not 1 |
| | 20 – 3 | division first, then subtraction |
| 2. | 10 + 8/2 | equals 14 not 9 |
| | 10 + 4 | division first then addition |
| 3. | 5 * 4/2 | equals 10 |
| | 20/2 | division and multiplication are equal |
| 4. | (20 – 15)/5 | does equal 1 |
| | 5/5 | parentheses overrides order |
| 5. | (10 + 8)/2 | equals 9 |
| | 18/2 | parentheses overrides order |
| 6. | 6 + 2 > 3 | true |
| | 8 > 3 | |
| 7. | 2 > 1 + 10 | false |
| | 2 < 11 | |

8. 2 + 2/2 * 6 equals 8 not 18
   2 + 1 * 6 division first
   2 + 6 then multiplication
   8 then addition

These examples have all the elements of an arithmetic expression—numeric values and operators—but they are not complete EasyLanguage statements. An expression must be part of either an assignment statement (myValue = mySum + myTot) or a logical statement (cond1 = cond2 OR cond3).

The overall purpose of EasyLanguage is to translate an idea and perform an analysis on a price data series over a specific time period. A price chart consists of bars built from historical price data. Each individual bar is a graphical representation of the range of prices over a certain period of time. A five-minute bar would have the Opening, High, Low, and Closing prices of an instrument over a five-minute time frame. A daily bar would graph the range of prices over a daily interval. Bar charts are most often graphed in an Open, High, Low, and Close format. Sometimes the opening price is left off. A candlestick chart represents the same data, but in a different format. It provides an easier way to see the relationship between the opening and closing prices of a bar chart. Other bar data such as the date and time of the bar's close, volume, and open interest is also available for each bar. Since EasyLanguage works hand-in-hand with the charts that are created by TradeStation, there are many built-in reserved words to interface with the data. These reserved words were derived from commonly used verbiage in the trading industry. You can interface with the data by using the following reserved words. (Note: Each word has an abbreviation and can be used as a substitute.)

| **Reserved Word** | **Abbreviation** | **Description** |
|---|---|---|
| Date | D | Date of the close of the bar. |
| Time | T | Time as of the close of the bar. |
| Open | O | Open price of the bar. |
| High | H | High price of the bar. |
| Low | L | Low price of the bar. |
| Close | C | Close price of the bar. |
| Volume | V | Number of contracts/shares traded. |
| OpenInt | OI | Number of outstanding contracts (futures only). |

If you wanted to determine that the closing price of a particular instrument was greater than its opening price you would simply type: Close > Open, or, C > O.

The beauty of EasyLanguage is its ability to have all of the data of an instrument at your fingertips. The reserved words that we use to access the different prices of the current bar are also used to access historical data. You do this by adding an index to the reserved word. The closing price of yesterday would be: Close[1]. The closing price two days ago would be: Close[2] and so on. The number inside the bracket determines the number of bars to look back. The larger the number, the further you go back in history. If you wanted to compare today's closing price with the closing price ten days prior you would type: Close > Close[10].

Before we move on, we should discuss how TradeStation stores dates and times. January 1, 2001 is stored as 1010101 instead of 20010101 or 010101. When the millennium changed, instead of incorporating the century into the date, TradeStation simply added a single digit to the year. The day after 991231 was 1000101 according to TradeStation. Time is stored as military time. For example, one o'clock in the afternoon is 1300 and one o'clock in the morning is 100.

After that brief introduction to the world of programming, let's go ahead and set up and program a complete trading strategy. The PowerEditor is where all of the magic takes place. This is your interface between your ideas and their applications. All of your coding takes place here, and a thorough understanding of this editor will reduce headaches and increase productivity. In TradeStation 4.0 and 2000i, the PowerEditor is basically a stand-alone application; it is an independent program and can be run with or without TradeStation. In the newer TradeStation 6.0, the PowerEditor is more of a component program; it runs within the confines of TradeStation.

## TRADESTATION 2000i VERSUS TRADESTATION 6.0

As of the writing of this book, there are two versions of TradeStation currently being used: 2000i and 6.0. Based on input from users, we figure that half is using one version and the other half is using the other. For this reason, the remainder of this chapter will be broken into two main sections. (Beyond this chapter, however, we will include the EasyLanguage code for 2000i in Appendix B. We will concentrate on 6.0 in the remaining chapters as it is the latest version and seems to be the way of the future.) Version 6.0 and 2000i are different enough to merit treating each one separately and we will refer to the two versions as 6.0 and 2000i. TradeStation 6.0 is Omega Research's all-inclusive trading tool. Everything that you need to design, test, monitor, and execute an analysis technique is in one slick and complete package. Omega Research is now a brokerage/software company and equities and futures trades can be executed through direct access with TradeStation.

### *TradeStation 2000i*

*PowerEditor*

Once this program is up and running, go under the File menu and select New. A dialog box titled *New* will open. If the General tab is not selected, go ahead and select it. Your screen should look like this:

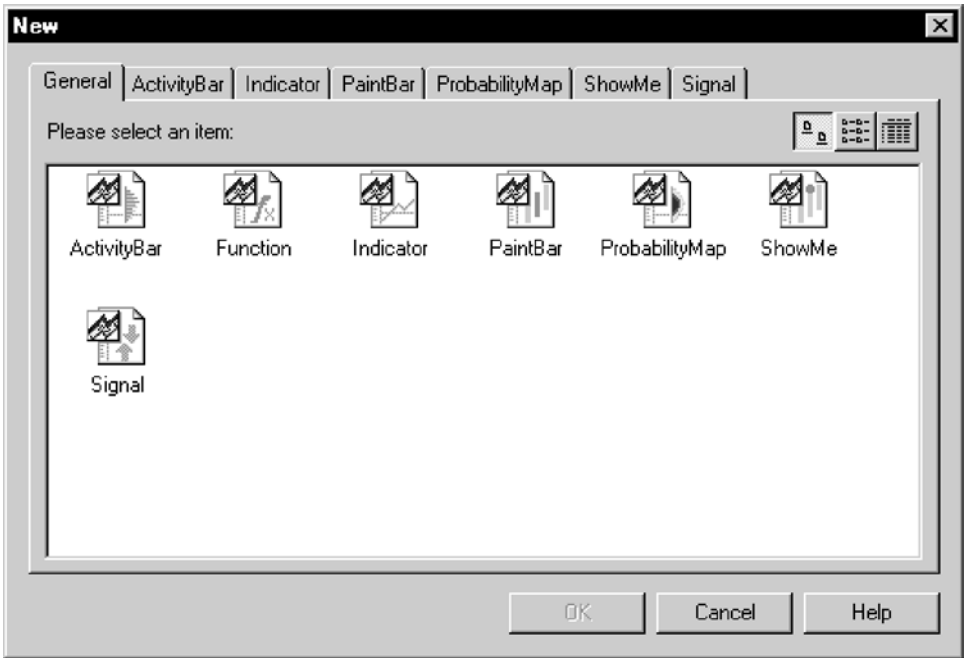


**Figure 1.1** New Dialog—TradeStation 2000i

Once your screen looks like Figure 1.1 select the icon with the title Signal and click OK. We will ignore the other tabs in this dialog box at the moment. Another dialog box titled *Create a New Signal* will open and ask for a name and notes about the signal that you are creating. In the Name field go ahead and type "MySignal-1" and in the Notes field type "Donchian Break Out" and then click OK. A window titled *MySignal-1* should open. This is your clean sheet of paper on which to type your wonderful ideas. Before we do some coding, let's briefly take a look at some of the menus. Table 1.1 details the selections that are available under the File menu.

Table 1.2 details the selections under the Edit menu.

Table 1.3 details the selections under the View menu.

Table 1.4 details the selections under the Tool menu.

**Table 1.1**
File Menu

| Menu Item | Action |
|---|---|
| New | Creates a new blank analysis technique. |
| Open | Opens an existing analysis technique. |
| Close | Closes current window. |
| Save | Saves the current analysis technique. |
| SaveAs | Allows renaming of the current analysis technique. |
| SaveAs Template | Saves the current analysis technique as a template for future use. |
| Save All | Saves all currently open analysis techniques. |
| Import and Export | Imports existing analysis techniques from other sources or exports analysis techniques to other sources. |
| Protect | Protects the analysis technique with a password. |
| Verify | Verifies the analysis technique. Checks for syntax errors in code. |
| Verify All | Verifies all functions and analysis techniques. |
| Properties | Shows the name and notes of the analysis technique. Allows the user to change these. |
| Page Setup | Allows changing of the page setup. |
| Print | Prints the code. |
| Print Preview | Shows what will be printed. |
| Exit | Exits out of PowerEditor. |

**Table 1.2**
Edit Menu

| Menu Item | Action |
|---|---|
| Undo | Undoes the last action |
| Redo | Redoes the last action. |
| Cut | Cuts the selected text. |
| Copy | Copies the selected text. |
| Paste | Pastes the selected text. |
| Clear | Clears the selected text. |
| Select All | Selects all text in the analysis technique. |
| Find | Finds a specific string of text. |
| Find Next | Finds the next occurrence of the specified string of text. |
| Find in Files | Powerful multifile search tool. |
| Replace | Replaces a string of text with another string of text. |

**Table 1.3**
View Menu

| Menu Item | Action |
|---|---|
| Toolbars | Customizes the tool bars. |
| Status Bar | Hides/shows the Status bar. |
| Output Bar | Hides/shows the Output bar. This little window will become important when we start debugging. |
| Bookmarks | Marks a particular section of text for reference purposes |
| Font | Sets the PowerEditor's font. |
| Options | Displays options for the PowerEditor. |

**Table 1.4**
Tool Menu

| Menu Item | Action |
|---|---|
| EasyLanguage Dictionary | Inserts EasyLanguage components into an analysis technique. |
| Errors Window Options | Changes the look of the Errors window. |
| Find In Files Window Options | Changes the look of the Find In Files window. |
| Debug Window | Changes the attributes of the Debug window. |

*A Simple Program*

Now that we are familiar with the menus and their functions, let's go ahead and code a simple program. We don't need to know everything about the selections in the menus to start coding. Jump in headfirst and type the following text exactly as you see it here:

```
Inputs: longLength(40), shortLength(40);

Buy tomorrow at Highest(High,longLength) stop;
Sell tomorrow at Lowest(Low,shortLength) stop;
```

(Note: for those of you who may be moving to TradeStation 6.0 you must type Sell Short to initiate a short position.)

After you have typed this, go under the File menu and select Verify or hit the F3 key. Many of the commands in the menus have keyboard equivalents or shortcuts. You can determine the shortcuts by selecting the menu and then the menu item. (The keyboard shortcut is listed to the far right of the menu item.) If you look at the Verify menu item on the File menu, you will see F3. You should get a small dialog box that first says *Verifying* and then *Excellent*. If you

get an error, simply check your code for any typos and Verify again. Congratulations, you have just written an analysis technique in the form of a signal! Now let's break each line of code down so that we can fully understand what is going on.

```
Inputs: longLength(40), shortLength(40);
```

By typing this line you have created two constant variables of the numeric data type. These two variables, longLength and shortLength, have been initiated with the value of 40. These variables cannot be changed anywhere in the body of the analysis technique. They can only be changed from the user interface of this signal or in the program heading. This will be discussed later in this chapter.

```
Buy tomorrow at Highest(High,longLength) stop;
```

This line instructs the computer to place a buy stop tomorrow at the highest high of the last 40 days. Highest is a function call. Functions are subprograms that are designed for a specific purpose and return a specific value. To communicate with a function, you must give it the information it needs. An automatic teller machine is like a function; you must give it your PIN before it will give you any money. In the case of the Highest function, it needs two bits of information: what to look at and how far to look back. We are instructing this function to look at the highs of the last 40 bars and return the highest of those highs. For now, just accept that High means High prices and Low means Low prices. (This is yet another subject that will be touched upon later.) When an order is instructed through EasyLanguage, you must tell the computer the type of order. In this case, we are using a stop order. Orders that are accepted by EasyLanguage are:

- *Stop*. Requires a price and is placed above the market to buy and below the market to sell.
- *Limit*. Requires a price and is placed below the market to buy and above the market to buy.
- *Market*. Buys/sells at the current market price.

So, if the market trades at a level that is equal to or greater than the highest high of the past forty days, the signal would enter long at the stop price.

```
Sell tomorrow at Lowest(Low,shortLength) stop;
```

The instructions for entering a short position are simply the opposite for entering a long position.